Лабораторна робота №6

Програмування побітових операцій

Мета: Навчитися програмувати на асемблері побітові операції, вивчити основні команди обробки бітів.

Завдання:

1. Створити у середовищі MS Visual Studio проект з ім’ям **Lab6**.
2. Написати вихідний текст програми згідно варіанту завдання. У проекті мають бути три модуля на асемблері:
   - головний модуль: файл **main6.asm**. Цей модуль створити та написати заново;
   - другий модуль: використати **module** попередніх робіт;
   - третій модуль: модуль **longop** попередньої роботи №5 доповнити новим кодом відповідно завданню.
3. У цьому проекті кожний модуль може окремо компілюватися.
4. Скомпілювати вихідний текст і отримати виконуємий файл програми.
5. Перевірити роботу програми. Налагодити програму.
6. Отримати результати – кодовані значення чисел згідно варіанту завдання.
7. Проаналізувати та прокоментувати результати, вихідний текст та дизасембльований машинний код програми.

Теоретичні відомості

Програмісту на асемблері потрібно враховувати, що внутрішнє представлення інформації та її обробка, зокрема, процесором, грунтується на двійкових кодах. Кожна команда так чи інакше змінює біти у регістрах або пам’яті. Проте можна виділити деякі команди, які прийнято називати командами "побітових", або порозрядних операцій. Вони надають програмісту широкі можливості маніпулювати окремими бітами або групами бітів даних.

Важливим також є те, що побітові команди відносяться до найшвидших команд для усіх цифрових процесорів. Кожний хто програмує на асемблері, повинен знати та вміти використовувати такі команди. Це також сприяє розробці та реалізації ефективних алгоритмів та програм.

Команди побітових операцій

Команда AND. Побітова кон'юнкція двійкових кодів двох операндів

```
and dest, src
```

Наприклад:

```
mov al, 75h          ; 01110101
and al, 3Eh          ; 00111110
                ; AL = 00110100
```

У деяких алгоритмах побітову кон'юнкцію використовують для виділення, "вирізання" окремих бітів. Для виділення окремого біту виконується побітове AND з відповідною "маскою" – двійковим кодом у якому потрібний біт дорівнює 1, а усі решта бітів – 0, наприклад:

```
and eax, 00004000h     ; "вирізання" 14-го біту маскою 0..0 0100 0000 0000 0000
and cx, 0008h          ; "вирізання" 3-го біту маскою 0..0 1000
and edx, 0F0000000h    ; "вирізання" чотирьох старших бітів у регістрі EDX
and ebx, 0FFFF7FFFh    ; обнулення  15-го біту у регістрі EBX маскою 1..101..1
```

Після того, як виконано операцію AND, наприклад, EAX із операндом-маскою **00004000h**, то, щоб знайти чому дорівнює окремий 14-й біт регістру EAX, порівнюємо EAX з нулем, наприклад, командою CMP:

```
cmp eax, 0
```

Якщо EAX дорівнює 0, то досліджуваний 14-біт є нульовим.

Команда OR. Побітова диз'юнкція двійкових кодів двох операндів

```
or dest, src
```

Наприклад:

```
mov al, 60h          ; 01100000
or al, 3Ah           ; 00111010
               ; AL = 01111010

or eax, 00008000h    ; 15-й біт регістру EAX стає 1, решта бітів не змінюється
```

Команда XOR. Нерівнозначність (*eXclusive OR*) бітів двійкових кодів двох операндів.

```
xor dest, src
```

Якщо біт першого операнду дорівнює відповідному біту другого операнду, то біт результату (операнду dest) буде 0, інакше – 1. Наприклад:

```
mov al, 75h          ; 01110101
xor al, 3Eh          ; 00111110
                ;AL = 01001011

mov al, 75h          ; 01110101
xor al, 20h          ; 00100000
                ;AL = 01010101   інверсія 5-го біту

mov al, 75h          ; 01110101
xor al, 0FFh         ; 11111111
                ;AL = 10001010   інверсія усіх бітів регітру AL
```

Дуже популярним у багатьох програмах є використання команди XOR для обнулення регістрів, наприклад:

```
xor eax, eax          ; EAX = 0
```

це працює швидше, аніж

```
mov eax, 0
```

Команда NOT. Виконується побітова інверсія – нульові біти замінюються на 1, а одиничні стають 0.

Наприклад:

```
mov al, 75h          ; 01110101
not al               ; 10001010
```

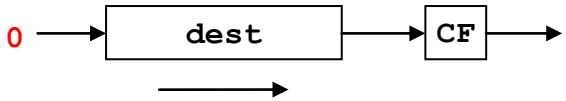
Команди зсуву

Такі команди зсувають біти двійкового коду операндів.

Команда SHR. Зсув бітів вправо (у напрямку молодшого біту).

```
shr dest, count
```

Виконується зсув бітів коду, записаного у операнді **dest**, вправо на **count** бітів. У старші **count** бітів записуються нулі.

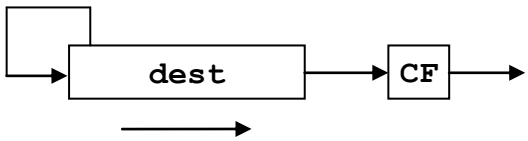


Операнд **dest** може вказувати регістр або адресу пам'яті. Операнд **count** може бути безпосереднім значенням або регістром CL. Значення операнду **count** у 32-бітовому режимі процесора може бути від 0 до 31.

Наприклад:

```
mov eax, 0F00B000h        ;00001111000000000101100000000000
shr eax, 7                ;00000000000011110000000101100000
```

Команда SAR. Зсув бітів вправо арифметичний. Старший (лівий) біт розмножується.



Якщо лівий біт операнду **dest** дорівнює 0, то зсув виконується так само, як і для команди SHR. Наприклад:

```
mov eax, 0F00B000h        ;00001111000000000101100000000000
sar eax, 7                ;00000000000011110000000101100000
```

Проте, якщо старший біт операнду є 1, то при зсуві розмножуються вже одиниці:

```
mov eax, -1580246784      ;10100001110011110101100100000000
sar eax, 7                ;11111110100001110011110101100100
```

Команди SAL, SHL. Зсув бітів вліво (у напрямку старшого біту).

```
shl dest, count
```

Виконується зсув бітів коду, записаного у операнді **dest**, вліво на **count** бітів. У молодші **count** бітів записуються нулі.

Операнд **dest** може вказувати регістр або адресу пам'яті. Операнд **count** може бути безпосереднім значенням або регістром CL. Значення операнду **count** у 32-бітовому режимі процесора може бути від 0 до 31.

Наприклад:

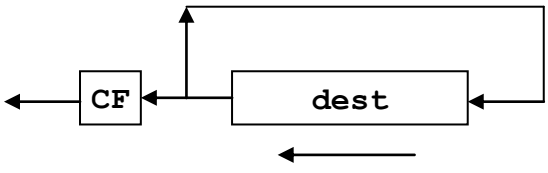
```
mov ax, 2Fh          ; 0000 0000 0010 1111
mov cl, 7            ; визначаємо зсув на 7 бітів
shl ax, cl           ; 0001 0111 1000 0000

mov eax, 2Fh         ; 0000 0000 0000 0000 0000 0000 0010 1111
shl eax, 13          ; 0000 0000 0000 0101 1110 0000 0000 0000

mov eax, 2Fh         ; 0000 0000 0000 0000 0000 0000 0010 1111
sal eax, 13          ; 0000 0000 0000 0101 1110 0000 0000 0000
```

Команди SAL та SHL працюють однаково.

Команди циклічного зсуву

Команда ROL. Циклічний зсув бітів вліво (у напрямку старшого біту). Старші біти записуються на молодші позиції.

```
rol dest, count
```

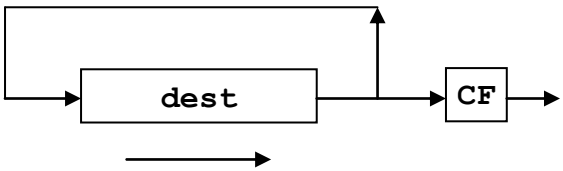


Наприклад:

```
mov ax, 7215h        ; 0111 0010 0001 0101
rol ax, 4            ; 0010 0001 0101 0111
```

Команда ROR. Циклічний зсув бітів вправо (у напрямку молодшого біту). Старші біти записуються на молодші позиції.

```
ror dest, count
```



Наприклад:

```
mov ax, 7215h        ; 0111 0010 0001 0101
ror ax, 4            ; 0101 0111 0010 0001
```

Приклади доступу до окремих бітів даних

Як узнати значення потрібного біту, наприклад, 67-го біту у 128-бітовому блоці даних? Вирішити це завдання можна наступним чином. Розглянемо блок даних як масив байтів (рис. 1).

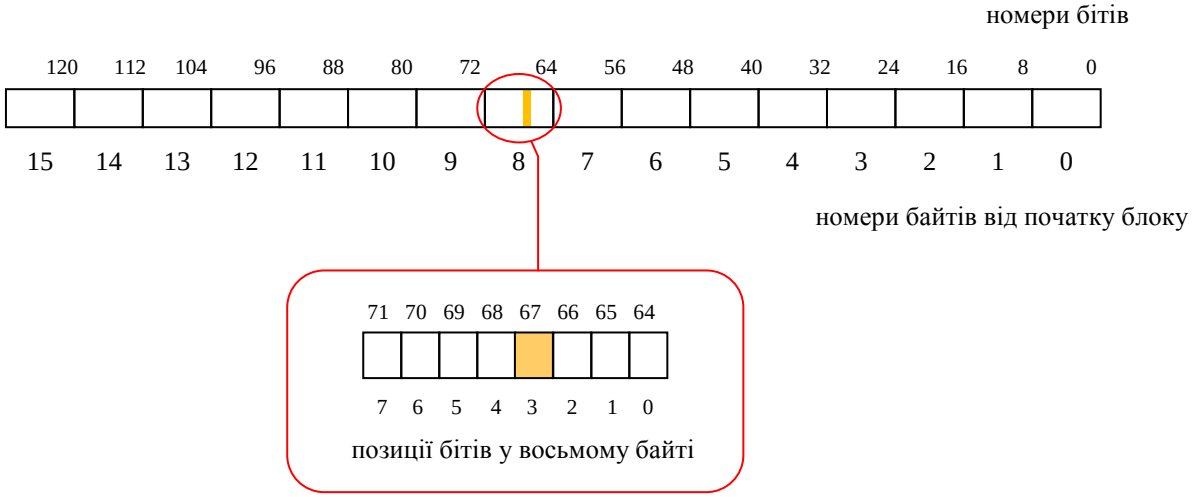


Рис. 1. Блок даних як масив байтів

У якому байті знаходиться потрібний біт? Номер байту дорівнює номеру біту, діленому на вісім: 67/8 = 8. Таким чином, 67-й біт знаходиться у 8-му байті.

Позиція потрібного біту є залишком від ділення номера біту на 8. Для 67-го біту це позиція 3.

Усе це запрограмувати можна, наприклад, так:

```
mov al, byte ptr [myData+8]      ; у регістр AL завантажуємо 8-й байт
and al, 08h                      ; "вирізаємо" 3-й біт:
                                 ;            AL = XXXX XXXX
                                 ;and        08h = 0000 1000
                                 ;                ---------
                                 ;результат: AL = 0000 X000
```

А як запрограмувати читання будь-якого довільного N-го біту? Для цього потрібно обчислювати номер байту та позицію біту. Ділення N на 8 та знаходження залишку від ділення можна виконати одною командою DIV, проте вона працює не швидко і є не зовсім зручною для вирішення нашого завдання.

Ділення на 8 – це те саме, що зсув на 3 біти вправо. Так можна запрограмувати обчислення номеру байту.

Знаходження залишку від ділення числа на 8 можна запрограмувати як вирізання трьох молодших бітів цього числа. Після знаходження залишку потрібно сформувати бітову маску – цю маску можна уявити як результат зсуву коду 00000001b вліво відповідно позиції потрібного біту.

Приклад реалізації наведеного вище алгоритму:

```
mov ebx, Nbit          ; Nbit – це номер біту
mov ecx, ebx
shr ebx, 3             ; номер байту

and ecx, 07h           ; бітова позиція = вирізаємо 3 молодші біти
mov al, 1
shl al, cl             ; AL = маска вирізання біту Nbit

mov ah, byte ptr [myData+ebx]
and ah, al             ; результат у регістрі AH
```

Значення Nbit-го біту міститься у регістрі AH, про що можна дізнатися так: якщо AH=0, то біт був нульовим, а якщо AH ≠ 0, то біт був 1.

Запис потрібного біту у блок даних. Спочатку так само можна знайти номер байту у блоці та бітову позицію у окремому байті. Далі, якщо треба встановити потрібний біт у 1, то виконується команда OR з 0..010..0. Якщо треба обнулити потрібний біт, то це можна зробити командою AND з 1..101..1. Наприклад:

```
  mov ah, 1              ; вказування потрібного значення біту

  mov ebx, Nbit          ; Nbit – це номер біту
  mov ecx, ebx
  shr ebx, 3             ; номер байту
  and ecx, 07h           ; позиція потрібного біту у байті
  mov al, 1
  shl al, cl             ; маска 0..010..0 за умовчанням

  cmp ah, 0
  jz @set0
  or byte ptr [myData+ebx], al
  jmp @goon

@set0:
  not al                 ; маска 1..101..1 для AND
  and byte ptr [myData+ebx], al

@goon:                   ; щось робимо далі
```

У цьому програмному коді потрібне значення Nbit-го біту від початку вказується у регістрі AH наступним чином: якщо AH=0, то Nbit-й біт блоку даних буде обнулятися, а якщо AH ≠ 0, то цей біт стане 1. Решта бітів блоку даних не змінюватиметься.

Передача числових значень та вказівників у параметрах процедур

Нехай потрібно передати у процедуру числове значення. Процедура повинна його сприйняти, щось виконати, та записати результат у потрібне місце пам'яті (наприклад, у якусь перемінну). Таким чином, у процедури можуть бути два параметри: один з них вона буде сприймати як просте числове значення, а другий параметр – як вказівник результату. По аналогії з мовою C/C++

```
void DoSomething(long *dest, long value);
```

Приклад реалізації на асемблері:

```
.data
  varA dd 2014
  varB dd ?
. . .

.code

; ця процедура має два параметри, які передаються через стек
DoSomething proc
  push ebp              ;пролог
  mov ebp, esp

  mov eax, [ebp+12]     ;перший параметр - звичайне число
  add eax, 8            ;якось його використовуємо

  mov ebx, [ebp+8]      ;другий параметр - вказівник
  mov [ebx], eax        ;запис значення EAX у пам'ять по вказівнику

  mov esp, ebp          ;епілог процедури - відновлюємо стек
  pop ebp
  ret 8                 ;у стеку були два параметри - 8 байтів
DoSomething endp
. . .

  push varA             ;перший параметр - значення перемінної varA
  push offset varB      ;другий параметр - адреса перемінної varB
  call DoSomething
```

Процедура DoSomething запише у перемінну varB значення 2022.

Для того, щоб у тілі процедури записати якійсь результат по адресі, на яку вказує параметр-вказівник, потрібно значення цього параметру спочатку завантажити у регістр, наприклад, EBX. А потім вже так:

```
mov [ebx], eax          ;запис значення EAX у пам'ять по вказівнику
```

або те саме, записане коректніше:

```
mov dword ptr[ebx], eax
```

Можливий універсальний підхід до вирішення завдань

Нижче пропонуються завдання запрограмувати виконання якихось побітових операції над бітовими даними (масивами) розрядністю до сотень бітів.

Безумовно, кожне завдання може вирішуватися якимось унікальним способом, який може бути специфічним для кожного варіанту завдання. Часто буває, що рішення є достатньо складним. Як саме вирішувати завдання – студент обирає індивідуально.

Але, можна запропонувати для вирішення багатьох (а, можливо, і для усіх) варіантів завдань деяку універсальну схему. У різних варіаціях цю схему можна використати для кожного варіанту завдань, причому можна відзначити достатню простоту та зрозумілість підходу, який пропонується. У якості прикладу можна розглянути зсув бітів на деяку кількість позицій. Універсальна схема у вигляді С-подібного псевдокоду може бути такою:

```
posSrc = стартова позиція біта звідки
posDest = стартова позиція біта куди
масив-копія = первісний масив        //копіювання, якщо потрібно
обнулення масиву результату          //обнулення, якщо потрібно
while (по усім бітам багаторозрядного масиву)
  {
  біт = ReadOneBit(posSrc)       //читаємо з масиву-копії
  WriteOneBit(біт)
  posSrc ++
  posDest ++
  }
```

Процедура **ReadOneBit** читає один біт, а процедура **WriteOneBit** – записує один біт так, як розглянуто вище у пп. Приклади доступу до окремих бітів даних.

Загальний підхід можна сформулювати так: у циклі читається по одному біту з одної позиції і записується цей біт (можливо, якось трансформований) у іншу позицію бітового поля.

Що конкретно виконується – визначається стартовими позиціями для бітів читання та запису. Наприклад, якщо **posSrc** = позиція старшого біта, а **posDest** = **posSrc** – 1, то фактично виконується зсув бітів праворуч (від молодших до старших), а якщо posSrc = 0, а posDest = M, то виконується зсув вліво на M бітів. І тому подібне.

Порядок виконання роботи та методичні рекомендації

1. Створіть у середовищі MS Visual Studio новий проект з ім’ям **Lab6**.

2. Додайте у проект порожній файл з ім’ям **main6.asm**. Цей файл буде головним файлом програмного коду.

3. Додайте у проект модулі **module** та **longop**. У проекті використовується файли **module.asm, module.inc, longop.asm**, **longop.inc** попередньої роботи №5.

4. Запрограмуйте у модулі **longop** процедуру обробки даних підвищеної розрядності згідно варіанту завдання. Назвіть процедуру ім’ям, наприклад, **Shr.** Рекомендується надати таке ім’я, яке позначає не тільки функцію, а й приналежність модулю, у якому ця процедура міститься, наприклад, **Shr_LONGOP**. Це дозволяє запобігати конфліктів імен у складних багатомодульних проектах.

5. У файлі **main6.asm** потрібно запрограмувати виклик процедури обробки даних підвищеної розрядності з тестовими значеннями параметрів та вивід результатів у шістнадцятковій системі числення у діалоговому вікні MessageBox. Запрограмувати вивід вихідних даних та результатів обробки у шістнадцятковому коді.

6. Компіляція, виклик програми, налагодження, отримання результатів. Виконання цих дій виконується у середовищі MS Visual Studio. Відомості та методичні рекомендації надані у відповідних розділах попередніх робіт.

Зміст звіту:

1. Титульний лист
2. Завдання
3. Роздруківка тексту програми
4. Роздруківка результатів виконання програми
5. Аналіз, коментар результатів, вихідного тексту та дизасембльованого машинного коду
6. Висновки

Варіанти завдань

Потрібно запрограмувати процедуру, яка обробляє дані підвищеної розрядності. У процедури мають бути такі параметри: адреса джерела даних, адреса результату, розрядність, а також (якщо потрібні) параметри N, M. Параметри N та M повинні бути довільними цілими. Якщо студент обмежить їх, наприклад, тільки кратними 8, то оцінка буде зменшуватися.

№ вар.	Операція	Розрядність (біт)
1	Зсув бітів вправо (подібно SHR) на N розрядів	256
2	Зсув бітів вправо арифметичний (подібно SAR) на N розрядів	288
3	Зсув бітів вліво (подібно SHL) на N розрядів	320
4	Зсув бітів вліво циклічний (подібно ROL) на N розрядів	352
5	Зсув бітів вправо циклічний (подібно ROR) на N розрядів	384
6	Обчислення кількості одиниць у двійковому коді	416
7	Обчислення кількості нулів у двійковому коді	448
8	Обчислення кількості старших нулів у двійковому коді	480
9	Обчислення кількості старших одиниць у двійковому коді	512
10	Запис M нулів починаючи з N-го біту. Решту бітів зробити 1	544
11	Запис M одиниць починаючи з N-го біту. Решту бітів зробити 0	576
12	Зсув вліво, щоб старший біт був 1 (нормалізація). Через аргумент	608
13	Зсув M старших бітів на N позицій вліво. Решта бітів нерухомі	640
14	Зсув M молодших бітів на N позицій вправо. Решта бітів нерухомі	672
15	Оберт бітів дзеркально відносно середньої позиції коду	704
16	Зсув вліво на N розрядів. У молодші N розрядів записується	736
17	Зсув вправо на N розрядів. У старші N розрядів записується	768
18	Починаючи з N-го розряду виконується побітове AND з M-бітовою	800
19	Починаючи з N-го розряду виконується побітове OR з M-бітовою	832
20	Починаючи з N-го розряду виконується побітове XOR з M-бітовою	864
21	Починаючи з N-го розряду виконується інверсія M бітів	896
22	Запис M бітів, починаючи з N-го розряду	928
23	З джерела даних, починаючи з N-го біту, взяти M бітів (M≤32) і	960
24	Обнулити M бітів, починаючи з N-го розряду	992
25	Запис 1 у M бітів, починаючи з N-го розряду	1024
26	Запис M одиниць починаючи з N-го біту. Решту бітів зробити 0	1056
27	Зсув бітів вліво циклічний (подібно ROL) на N розрядів	1088
28	Зсув бітів вправо циклічний (подібно ROR) на N розрядів	1120
29	Зсув бітів вправо арифметичний (подібно SAR) на N розрядів	1152
30	Обчислення кількості одиниць у двійковому коді	1184

Контрольні питання:

1. Що таке побітові операції?
2. Чим відрізняється команда SHR від SAL?
3. Який результат команди XOR і як вона може бути використана?
4. Яка команда виконує інверсію бітів?
5. Як можна узнати значення окремого біту?
6. Як можна записати кудись окремий біт?
7. Як зберігати та обробляти дані підвищеної розрядності?
8. Як запрограмувати параметри процедури?
9. Як запрограмувати запис результату через параметр процедури?
10. Як передати процедурі для обробки дані підвищеної розрядності?